Modern Software Design Methods for Concurrent and Real-Time Systems

Hassan Gomaa

1. INTRODUCTION

With the massive reduction in the cost of microprocessor and semiconductor chips and the large increase in microprocessor performance over the last few years, real-time and distributed real-time microcomputer-based systems are a very cost-effective solution to many problems. Nowadays, more and more commercial, industrial, military, medical, and consumer products are microcomputer based and either software controlled or have a crucial software component to them. This paper presents an overview of the design of concurrent systems, as well as an important category of concurrent systems, namely real-time systems.

A characteristic of all real-time systems is that of concurrent processing; that is, many activities occurring simultaneously whereby, frequently, the order of incoming events is not predictable. Consequently, as real-time systems deal with several concurrent activities, it is highly desirable for a real-time system to be structured into concurrent tasks (also known as concurrent processes).

This paper starts by providing an overview of concurrent processing concepts in Section 2. In Section 3, run-time support for concurrent and real-time systems is briefly discussed. Section 4 presents an overview of concurrent and real-time design methods. With this background, an overview of a modern design method for concurrent systems, including distributed and real-time systems, is given in Section 5. The COMET method [Gomaa 2000] integrates object-oriented and concurrent processing concepts, and uses the Unified Modeling Language (UML) notation.

2. CONCURRENT PROCESSING CONCEPTS

This section describes the concepts of the concurrent task (process), and the communication and synchronization between cooperating tasks. For more information, refer to Andrews 1991, Bacon 1997, Magee and Kramer 1999, Silberschatz and Galvin 1998, and Tanenbaum 1992.

2.1. Concurrent Tasks

A concurrent task (also known as a concurrent process) represents the execution of a sequential program or sequential component of a concurrent program. A concurrent system consists of several tasks executing in parallel. Each task deals with one sequential thread of execution. Concurrency in a software system is obtained by having multiple asynchronous tasks, running at different speeds. From time to time, the tasks need to communicate and synchronize their operations with each other. The concurrent tasking concept has been applied extensively to the design of operating systems, real-time systems, interactive systems, distributed systems, parallel systems, and simulation applications [Bacon 1997].

Consider the case of a robot system that controls up to four robot arms. Typically, there is one robot program to control the operations of one robot arm. Thus, each executing robot program is considered a software task.

In most concurrent systems, the tasks need to coordinate their activities. Two ways in which this procedure can be achieved are *mutual exclusion* and *task synchronization.*

2.2. Mutual Exclusion

Mutual exclusion is required when only one task at a time may have exclusive access to a resource. The part of a program that a task executes when it has exclusive access to a resource is referred to as its critical section or critical region. An application

Part of this material contained in this article was extracted from H. Gomaa, *Designing Concurrent, Distributed and Real-Time Applications with UML.* Addison-Wesley Object Technology Series, ISBN 0-201-65793-7, 2000 © 2000 Hassan Gomaa. Reprinted by permission of Addison-Wesley Longman.

of mutual exclusion is robot entry into a "collision zone." Collision could occur if more than one robot is allowed to move into the collision zone at the same time. To prevent this, robot entry into the collision zone is made mutually exclusive. For example, four robot tasks are performing independent assembly operations. However, there is a collision zone where robot arms could potentially collide.

The mutual exclusion problem may be solved by using semaphores [Andrews 1991, Bacon 1997]. Two operations are allowed on binary semaphores, P(s) and V(s), where (s) is the semaphore. The P operation is a potential wait operation. This operation is executed by a task when it wishes to enter the critical section. If the semaphore is set to 1 (meaning that no task is in the critical section), then the semaphore is decremented and the task is allowed to enter the critical section. If the semaphore is set to 0 when the P operation is executed by task A, this setting means that another task, say B, is in the critical section. Task A is suspended until task B signals that it is leaving the critical section by performing a V operation. This results in task A being allowed to enter the critical section. An example of this approach for a robot task that wishes to enter the collision zone follows:

```
Perform operations outside collision zone.

P (Collision_Zone_Semaphore)
Perform mutually exclusive operations in collision zone
V (Collision_Zone_Semaphore)

Perform more operations outside collision zone.
```

2.3. Synchronization of Tasks

Task Synchronization is required when one task, typically a producer, wishes to signal another, the consumer, to notify it that some event has occurred. For example, the producer robot moves a part into position and then signals the event Part_Ready. The consumer robot, which is suspended waiting for the signal, is reactivated so that it can move to the part and pick it up.

Events may be used to synchronize the operations of two tasks. The producer task can perform a signal function as follows: Signal (E), which signals that an event (E) has taken place. The consumer task may perform a Wait (E), which suspends the task until the event has been signaled by the producer. If the event has already been signaled, then the task is not suspended.

An example of using events for synchronizing two robot systems is now given. A pick-and-place robot is to bring a part to a robot that is to drill four holes in it. On completion, the pick-and-place robot is to move the part away.

The pick-and-place robot A moves the part into position and then signals the event Part_Ready. The drilling robot B is awakened, moves to the workplace and drills the holes. On completion, Robot B signals a second event, Part_Completed, which is waited on by A. After being awakened, Robot A removes the part. Each robot program is in a loop, as the robots will repetitively perform their tasks. The solution is as follows:

```
Robot A

WHILE Work_Available DO

Pick up part
Move Part to Workplace
Release part
Move to safe position
Signal (Part_Ready)
Wait (Part_Completed)
Pick up part
Remove from Workplace
END

Robot B:

WHILE Work_Available DO

Wait (Part_Ready)
Move to Workplace
Drill Four holes
Move to safe position
Signal (Part_Completed)
END
```

2.4. Message Communication

Concurrent tasks may communicate with each other using messages, which provide a general method of intertask communication [Andrews 1991, Bacon 1997]. Message communication is used when data needs to be passed between two tasks. Thus Signal and Wait event commands may be packaged into messages. For example, the producer task P sends a message M to the consumer C. The consumer task C may request to receive a message from the producer. If a message is already available, the consumer will receive it and continue processing; otherwise, the consumer is suspended until the message arrives.

Two types of message communication are possible: loosely couple message communication and tightly couple message communication. With loosely coupled message communication (also referred to as asynchronous message communication), the producer sends the message to the consumer and then continues processing. In this case, a message queue could build up between the producer and the consumer. There are two forms of tightly coupled message communication (also referred to as synchronous message communication): tightly coupled message communication with reply and tightly coupled message communication without reply. In the former case, the producer sends the message and then waits for a reply. When the consumer receives the message, it will generate a reply and send it back to the producer. Both producer and consumer then continue processing. With tightly coupled message communication without reply, the producer sends the message and then waits for acceptance of the message by the consumer. Tightly coupled message communication is known in Ada as the *rendezvous*. An example of message communication is that of data that needs to be passed between robots, or between a robot and another industrial system. For example, a vision system could pass part location data to a robot task.

As an example of message communication, consider the case in which a vision system has to inform a robot whether the type of car body frame coming down a conveyor is a sedan or station wagon. The robot has a different program for each car body type. In addition, the vision system has to send the robot information about the location and orientation of the car body frame on the conveyor. Usually, this information is sent as an offset (i.e., relative position) from a point known to both systems, with the part identification and offset information being sent in a message from the vision system to the robot. A sensor is used to indicate to the vision system that the car has arrived, which sets the external event signal Car_arrived. The robot indicates that it has completed the welding operations it performs on the car by signaling the actuator Move_car, which results in the car being moved away by the conveyor.

```
Vision System:

Wait (Car_arrived)
Take picture of car body
Identify car body
Determine location and orientation of car body
Send message (car model i.d., car body offset) to robot

Robot task:

Wait for message from vision system
Read message (car model i.d., car body offset)
Select welding program for car model
Execute welding program using offset for car position
Signal (Move_car)
```

3. RUN-TIME SUPPORT FOR CONCURRENT TASKS

Run-time support for concurrent processing may be provided by:

- **Kernel of an operating system.** This has the functionality to provide services for concurrent processing. In some modern operating systems, a microkernel provides minimal functionality to support concurrent processing, with most services provided by system-level tasks.
- **Run-time support system** for a concurrent language.
- **Threads package.** Provides services for managing threads (lightweight processes) within heavyweight processes.

3.1. Language Support for Concurrent Tasks

With sequential programming languages, such as C, C++, Pascal, and Fortran, there is no support for concurrent tasks. To develop a concurrent multitasked application using a sequential programming language, it is, therefore, necessary to use a kernel or threads package.

With concurrent programming languages, such as Ada and Java, the language supports constructs for task communication and synchronization. In this case, the language's runtime system handles task scheduling and provides the services and underlying mechanisms to support intertask communication and synchronization.

3.2. Real-Time Operating Systems

Much of the operating system technology for concurrent systems is also required for real-time systems. Most real-time operating systems support a kernel or microkernel, as described previously. However, real-time systems have special needs, many of which relate to having predictable behavior. It is more useful to consider the requirements of a real-time operating system than to provide an extensive survey of available real-time operating systems, because the list changes on a regular basis. Thus, a real-time operating system must:

- Support multitasking.
- Support priority preemption scheduling. This means each task needs to have its own priority.
- Provide task synchronization and communication mechanisms.
- Provide a memory-locking capability for tasks. In hard real-time systems, it is usually the case that all concurrent tasks are memory resident. This is to eliminate the uncertainty and variation in response time introduced by paging overhead. This memory-locking capability allows all time-critical tasks with hard deadlines to be locked in main memory so they are never paged out.
- Provide a mechanism for priority inheritance. When a task, task A, enters a critical section, its priority must be raised. Otherwise, task A is liable to get preempted by a higher-priority task, which is then unable to enter its critical section because of task A; hence, the higher priority task is blocked indefinitely.
- Have a predictable behavior (for example, for task context switching, task synchronization, and interrupt handling). Thus, there should be a predictable maximum response time under all anticipated system loads.

4. SURVEY OF DESIGN METHODS FOR CONCURRENT AND REAL-TIME SYSTEMS

For the design of concurrent and real-time systems, a major contribution came in the late seventies with the introduction of the MASCOT notation [Simpson 1979] and later the MASCOT design method [Simpson 1986]. Based on a data flow approach, MASCOT formalized the way tasks communicate with each other, via either channels for message communication or pools (information hiding modules that encapsulate shared data structures).

The 1980s saw a general maturation of software design methods, and several system design methods were introduced. Parnas's work with the Naval Research Lab, in which he explored the use of information hiding in large-scale software design, led to the development of the Naval Research Lab (NRL) Software Cost Reduction Method [Parnas, Clements, and Weiss 1984]. Work on applying structured analysis and structured design to concurrent and real-time systems led to the development of Real-Time Structured Analysis and Design (RTSAD) [Ward and Mellor 1985, Hatley and Pirbhai 1988] and the Design Approach for Real-Time Systems (DARTS) [Gomaa 1984] methods.

Another software development method to emerge in the early 1980s was Jackson System Development (JSD) [Jackson 1983]. JSD was one of the first methods to advocate that the design should model reality first and, in this respect, predated the object-oriented analysis methods. The system is considered a simulation of the real world and is designed as a network of concurrent tasks, where each real-world entity is modeled by means of a concurrent task. JSD also defied the then-conventional thinking of top-down design by advocating a middle-out behavioral approach to software design. This approach was a precursor of object interaction modeling, an essential aspect of modern object-oriented development.

The early object-oriented analysis and design methods emphasized the structural aspects of software development through information hiding and inheritance but neglected the dynamic aspects. A major contribution by the Object Modeling Technique [Rumbaugh et al. 1991] was to clearly demonstrate that dynamic modeling was equally important. In addition to introducing the static modeling notation for the object diagrams, OMT showed how dynamic modeling could be performed with state charts (hierarchical state transition diagrams originally conceived by Harel [Harel 1998, Harel et al. 1998]) for showing the state-dependent behavior of active objects and with sequence diagrams to show the sequence of interactions between objects.

The CODARTS (Concurrent Design Approach for Real-Time Systems) method [Gomaa 1993] built on the strengths of earlier concurrent design, real-time design, and early object-oriented design methods. These included Parnas's NRL Method, Booch Object-Oriented Design [Booch 1994], JSD, and the DARTS method by emphasizing both information hiding module structuring and task structuring. In CODARTS, concurrency and timing issues are considered during task design and information hiding issues are considered during module design.

Octopus [Awad, Kuusela, and Ziegler 1996] is a real-time design method based on use cases, static modeling, object interactions, and statecharts. By combining concepts from Jacobson's use cases with Rumbaugh's static modeling and statecharts, Octopus anticipated the merging of the notations that is now the UML. For real-time design, Octopus places particular emphasis on interfacing to external devices and on concurrent task structuring.

ROOM (Real-Time Object-Oriented Modeling) [Selic, Gullekson, and Ward 1994] is a real-time design method that is closely tied in with a CASE (Computer Assisted Software Engineering) tool called ObjecTime. ROOM is based on actors, which are active objects that are modeled using a variation on state charts called ROOMcharts. A ROOM model, which has been specified in sufficient detail, may be executed. Thus, a ROOM model is operational and may be used as an early prototype of the system.

Buhr and Casselman [1996] introduced an interesting concept called the use case map (based on the use case concept) to address the issue of dynamic modeling of large-scale systems. Use case maps consider the sequence of interactions between objects (or aggregate objects in the form of subsystems) at a larger-grained level of detail than do collaboration diagrams.

For UML-based real-time software development, Douglass [1999, 2004] has provided a comprehensive description of how UML can be applied to real-time systems. His 2004 book describes applying the UML notation to the development of real-time systems. The 1999 book is a detailed compendium covering a wide range of topics in real-time system development, including safety-critical systems, interaction with real-time operating systems, real-time scheduling, behavioral patterns, real-time frameworks, debugging, and testing.

5. A MODERN SOFTWARE DESIGN METHOD FOR CONCURRENT AND REAL-TIME SYSTEMS

5.1. Introduction

Most books on object-oriented analysis and design only address the design of sequential systems or omit the important design issues that need to be addressed when designing real-time and distributed applications [Gomaa 1993, Bacon 1997]. It is essential to blend object-oriented concepts with the concepts of concurrent processing in order to successfully design these applications. This paper describes some of the key aspects of the COMET method for designing real-time and distributed applications. COMET integrates object-oriented and concurrent processing concepts and uses the Unified Modeling Language (UML) notation (see article on Unified Modeling Language). It also describes the decisions made on how to use the UML notation to address the design of large-scale concurrent, distributed, and real-time applications. Examples are given from a pump monitoring and control system.

5.2. The COMET Method

COMET is a concurrent object modeling and architectural design method for the development of concurrent applications, in particular distributed and real-time applications [Gomaa 2000]. As the UML is now the standardized notation for describing object-oriented models [Booch et al. 1999, Rumbaugh et al. 2004, Jacobson et al. 2000], the COMET method uses the UML notation throughout.

The COMET object-oriented software life cycle is highly iterative. In the requirements modeling phase, a use case model is developed in which the functional requirements of the system are defined in terms of actors and use cases.

In the analysis modeling phase, static and dynamic models of the system are developed. The static model defines the structural relationships among problem domain classes. Object structuring criteria are used to determine the objects to be considered for the analysis model. A dynamic model is then developed in which the use cases from the requirements model are refined to show the objects that participate in each use case and how they interact with each other. In the dynamic model, state-dependent objects are defined using state charts.

In the design modeling phase, an architectural design model is developed. Subsystem structuring criteria are provided to design the overall software architecture. For distributed applications, a component-based development approach is taken, in which each subsystem is designed as a distributed, self-contained component. The emphasis is on the division of responsibility between clients and servers, including issues concerning the centralization versus distribution of data and control, and the design of message communication interfaces, including synchronous, asynchronous, brokered, and group communication. Each concurrent subsystem is then designed in terms of active objects (tasks) and passive objects. Task communication and synchronization interfaces are defined. The performance of real-time designs is estimated using an approach based on rate monotonic analysis [SEI 1993].

Distinguishing features of the COMET method are the emphasis on:

- Structuring criteria to assist the designer at different stages of the analysis and design process: subsystems, objects, and concurrent tasks.
- Dynamic modeling, both object collaboration and state charts, describing in detail how object collaborations and statec harts relate to each other.
- Distributed application design, addressing the design of configurable distributed components and intercomponent message communication interfaces.
- Concurrent design, addressing in detail task structuring and the design of task interfaces.
- Performance analysis of real-time designs using real-time scheduling.

COMET emphasizes the use of structuring criteria at different stages in the analysis and design process. Object structuring criteria are used to help determine the objects in the system, subsystem structuring criteria are used to help determine the subsystems, and concurrent task structuring are used to help determine the tasks (active objects) in the system. UML stereotypes are used throughout to clearly show the use of the structuring criteria.

The UML notation supports requirements, analysis, and design concepts. The COMET method separates requirements, analysis, and design activities. Requirements modeling addresses defining the functional requirements of the system. COMET differentiates analysis from design as follows. Analysis is breaking down or decomposing the problem so that it is understood better. Design is synthesizing or composing (putting together) the solution. These activities are now described in more detail.

5.3. Requirements Modeling with UML

In the requirements model, the system is considered as a black box. A use case model is developed in which the functional requirements of the system are defined in terms of use cases and actors. This section describes the use of actors in real-time applications.

There are several variations on how actors are modeled [Jacobson 1992, Booch 1998, Fowler 1999, Schneider and Winters 1998]. An actor is very often a human user. In many information systems, humans are the only actors. It is also possible in information systems for an actor to be an external system. In real-time and distributed applications, an actor can also be an external I/O device or a timer. External I/O devices and timer actors are particularly prevalent in real-time embedded systems, in which the system interacts with the external environment through sensors and actuators.

A human actor may use various I/O devices to physically interact with the system. In such cases, the human is the actor and the I/O devices are not actors. In some case, however, it is possible for an actor to be an I/O device. This can happen when a use case does not involve a human, as often happens in real-time applications.

An actor can also be a timer that periodically sends timer events to the system. Periodic use cases are needed when certain information needs to be output by the system on a regular basis. This is particularly important in real-time systems, although it can also be useful in information systems. Although some methodologists consider timers to be internal to the system, it is more useful in real-time application design to consider timers as logically external to the system and to treat them as primary actors that initiate actions in the system. An example of a use case model from the pump monitoring and control system is given in Figure 1.

5.4. Analysis Modeling with UML

This section describes some of the interesting aspects of COMET for analysis modeling. In particular, this section describes static modeling of the system context, stereotypes to represent object structuring decisions made by the analyst, and consistency checking between multiple views of a dynamic model.

5.4.1. Static Modeling

For real-time applications, it is particularly important to understand the interface between the system and the external environment, which is referred to as the *system context.* In structured analysis [Yourdon 1989], the system context is shown on a *system context diagram.* The UML notation does not explicitly support a system context diagram. However, the system context may be depicted using either a static model or a collaboration model [Douglass 1999]. A *system context class diagram* provides a more detailed view of the system boundary than a use case diagram.

Using the UML notation for the static model, the system context is depicted showing the system as an aggregate class with the stereotype «system», and the external environment is depicted as external classes to which the system has to interface (Figure 2). External classes are categorized using stereotypes. An external class can be an «external input device», an «external output device», an «external I/O device», an «external user», an «external system», or an «external timer». For a real-time

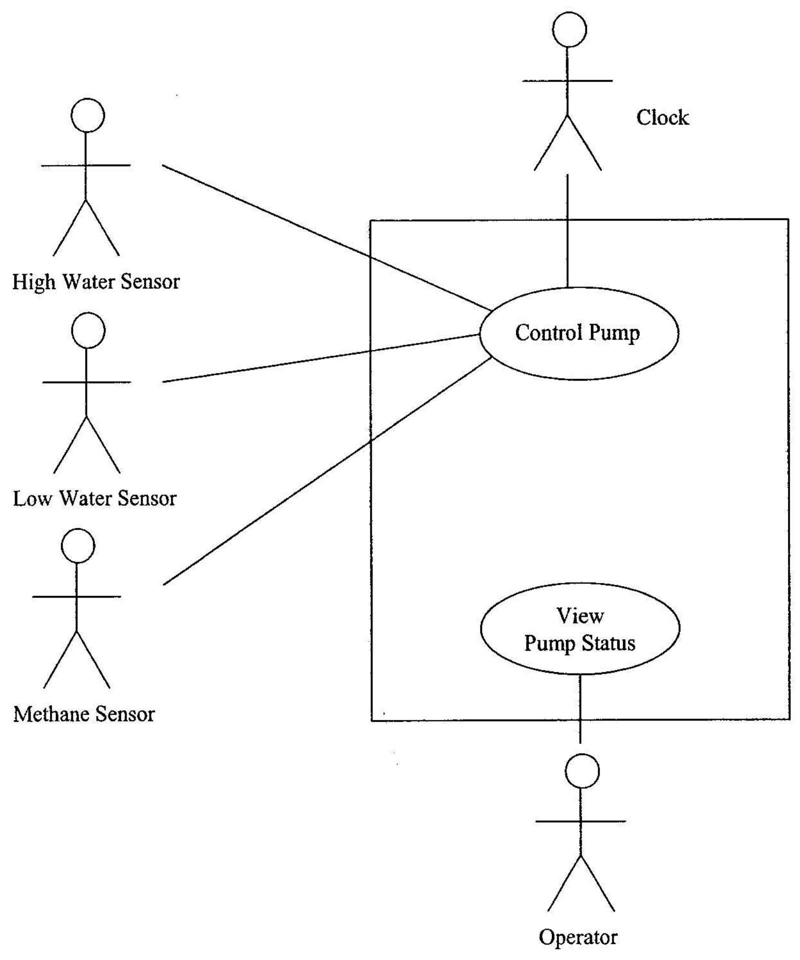

Figure 1. Use case model for pump monitoring and control system.

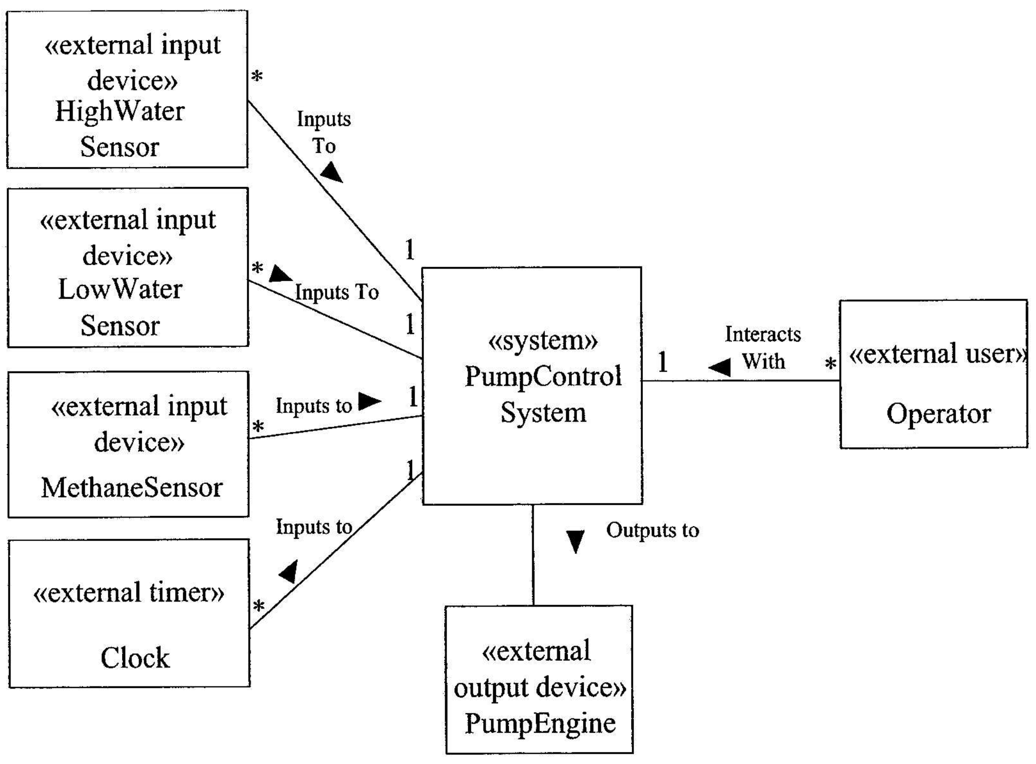
«external input device» HighWater Sensor
«external input device» LowWater Sensor
«external input device» MethaneSensor
«external timer» Clock
«system» PumpControl System
«external user» Operator
«external output device» PumpEngine
Inputs To
Inputs To
Inputs to
Inputs to
Interacts With
Outputs to
*
1

Figure 2. Pump monitoring and control system class context diagram.

system, it is desirable to identify low-level external classes that correspond to the physical I/O devices with which the system has to interface. These external classes are depicted with the stereotype «external I/O device».

An example of a system context class diagram from the pump Monitoring and control system is given in Figure 2.

During the analysis modeling phase, static modeling is also used for modeling data-intensive classes [Rumbaugh 1991].

5.4.2. Object Structuring

Object structuring criteria are provided to assist the designer in structuring a system into objects. Several object-based and object-oriented analysis methods provide criteria for determining objects in the problem domain [Booch 1994, Coad and Yourdon 1991, Gomaa 1993, Jacobson 1992, Parnas et al. 1984, Shlaer and Mellor 1988]. The COMET object structuring criteria build on these methods.

In object structuring, the goal is to categorize objects in order to group together objects with similar characteristics. Whereas classification based on inheritance is an objective of object-oriented modeling, it is essentially tactical in nature. Categorization, however, is a strategic classification. It is a decision to organize classes into certain groups because most software systems have these kinds of classes and categorizing classes in this way helps us understand the system we are to develop.

UML stereotypes are used to distinguish among the different kinds of application classes. A *stereotype* is a subclass of an existing modeling element, in this case an application class, which is used to represent a usage distinction, in this case the kind of class. A stereotype is depicted in guillemets, e.g., «interface». An instance of a stereotype class is a stereotype object, which can also be shown in guillemets. Thus, an application class can be categorized as an «entity» class, which is a persistent class that stores data; an «interface» class, which interface to the external environment; a «control» class, which provides the overall coordination for the objects that participate in a use case; or an «application logic» class, which encapsulates algorithms separately from the data being manipulated.

Real-time systems will have many device interface classes to interface to the various sensors and actuators. They will also have complex, state-dependent control classes because these systems are highly state dependent.

5.4.3. Dynamic Modeling

For concurrent, distributed, and real-time applications, dynamic modeling is of particular importance. UML does not emphasize consistency checking between multiple views of the various models. Nevertheless, during dynamic modeling, it is important to understand how the finite-state machine model, depicted using a state chart [Harel 1988, Harel and Gary 1996, Harel and Politi 1998] that is executed by a state-dependent control object, and relates to the interaction model, which depicts the interaction of this object with other objects.

State-dependent dynamic analysis addresses the interaction among objects that participate in state-dependent use cases. A state-dependent use case has a state-dependent control object, which executes a state chart, providing the overall control and sequencing of the use case. The interaction among the objects that participate in the use case is depicted on a collaboration diagram or sequence diagram.

The state chart needs to be considered in conjunction with the collaboration diagram. In particular, it is necessary to consider the messages that are received and sent by the control object, which executes the state chart. An input event into the control object on the collaboration diagram must be consistent with the same event depicted on the state chart. The output event (which causes an action, enable or disable activity) on the state chart must be consistent with the output event shown on the collaboration diagram.

An example of the collaboration diagram for the control pump use case is given in Figure 3, and the state chart for the Pump Control object is shown in Figure 4.

5.5. Design Modeling

This section describes some of the interesting aspects of COMET for design modeling. In particular, this section describes the consolidation of collaboration diagrams to synthesize an initial software design, subsystem structuring using packages, distributed application design, concurrent task design, and the design of connectors using monitors.

5.5.1. The Transition from Analysis to Design

In order to transition from analysis to design, it is necessary to synthesize an initial software design from the analysis carried out so far. In the analysis model, a collaboration diagram is developed for each use case. The *consolidated collaboration diagram* is a synthesis of all the collaboration diagrams developed to support the use cases. The consolidation performed at this stage is analogous to the robustness analysis performed in other methods [Jacobson 1992, Rosenberg and

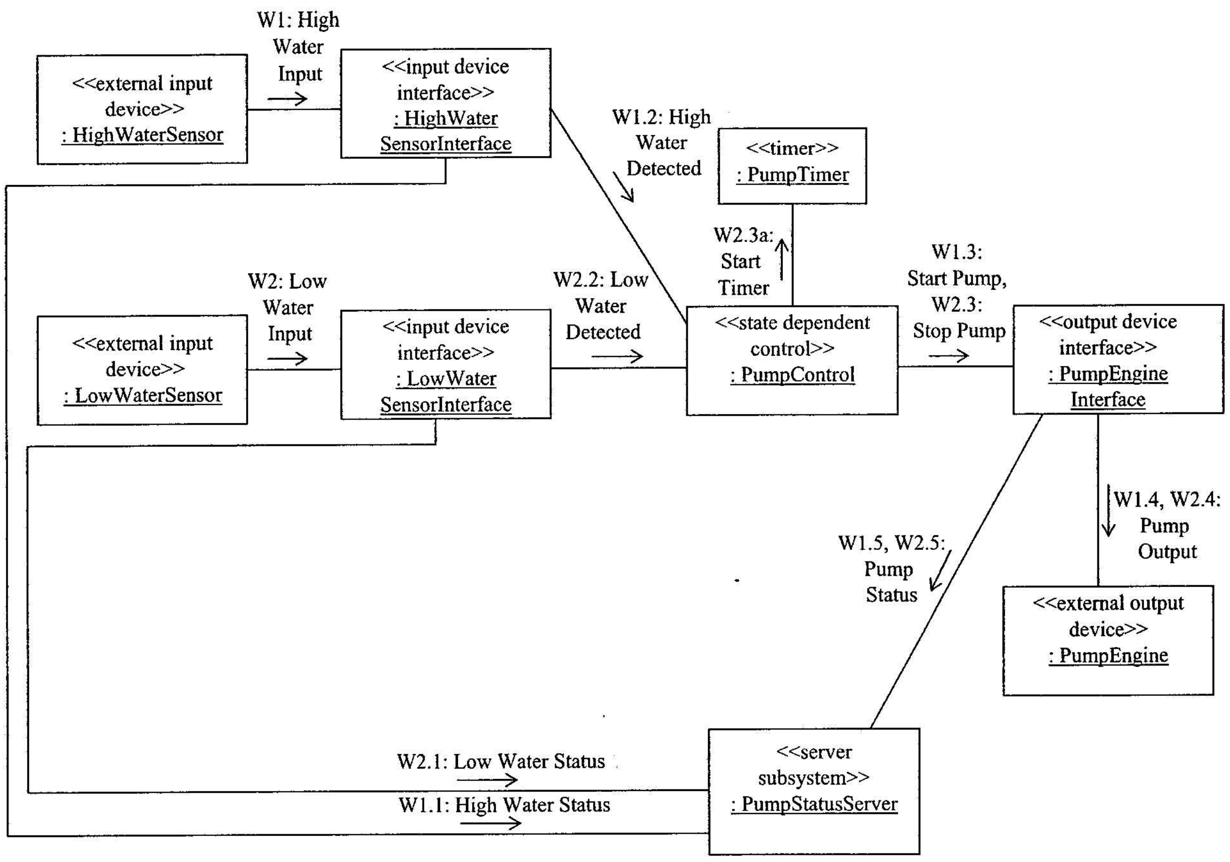
W1: High Water Input
<<external input device>> : HighWaterSensor
<<input device interface>> : HighWater SensorInterface
W1.2: High Water Detected
<<timer>> : PumpTimer
W2.3a: Start Timer
W2: Low Water Input
<<external input device>> : LowWaterSensor
<<input device interface>> : LowWater SensorInterface
W2.2: Low Water Detected
<<state dependent control>> : PumpControl
W1.3: Start Pump, W2.3: Stop Pump
<<output device interface>> : PumpEngine Interface
W1.4, W2.4: Pump Output
<<external output device>> : PumpEngine
W1.5, W2.5: Pump Status
W2.1: Low Water Status
W1.1: High Water Status
<<server subsystem>> : PumpStatusServer

Figure 3. Collaboration diagram for control pump use case.

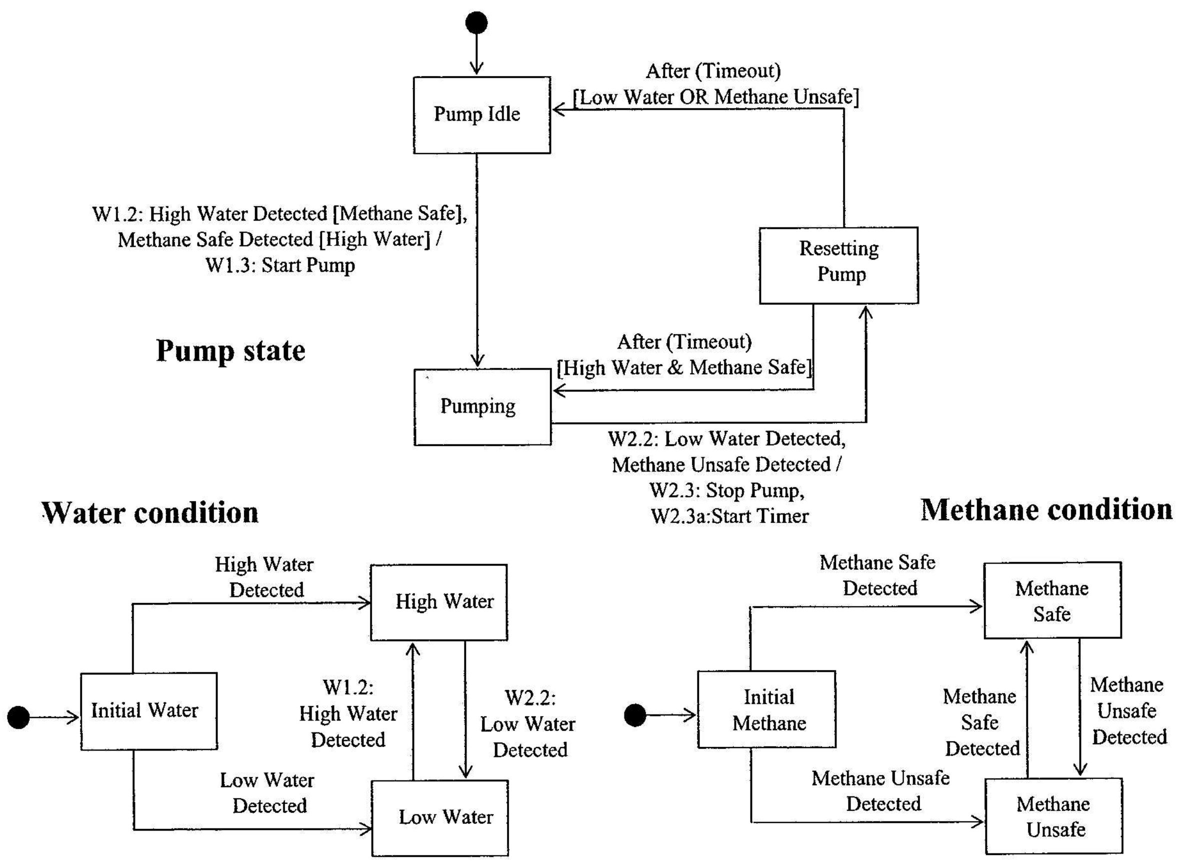
Pump state
Pump Idle
After (Timeout) [Low Water OR Methane Unsafe]
W1.2: High Water Detected [Methane Safe], Methane Safe Detected [High Water] / W1.3: Start Pump
Resetting Pump
After (Timeout) [High Water & Methane Safe]
Pumping
W2.2: Low Water Detected, Methane Unsafe Detected / W2.3: Stop Pump, W2.3a:Start Timer
Water condition
High Water Detected
High Water
Initial Water
W1.2: High Water Detected
W2.2: Low Water Detected
Low Water Detected
Low Water
Methane condition
Methane Safe Detected
Methane Safe
Initial Methane
Methane Safe Detected
Methane Unsafe Detected
Methane Unsafe Detected
Methane Unsafe

Figure 4. Pump control state chart.

Scott 1999]. These other methods use the static model for robustness analysis, whereas COMET emphasizes the dynamic model, as this addresses the message communication interfaces, which are crucial in the design of real-time and distributed applications.

The consolidated collaboration diagram, which depicts the objects and messages from all the use-case-based collaboration diagrams, can get very large for a large system and thus it may not be practical to show all the objects on one diagram. This problem is addressed by developing a consolidated collaboration diagram for each subsystem, and developing a higher-level subsystem collaboration diagram to show the dynamic interactions between subsystems on a *subsystem collaboration diagram,* which depicts the overall software architecture, as shown in Figure 5. The structure of an individual subsystem is then depicted on a consolidated collaboration diagram, which shows all the objects in the subsystem and their interconnections.

5.5.2. Software Architecture Design

During software architecture design, the system is broken down into subsystems and the interfaces between the subsystems are defined [Shaw and Garlan 1996]. A system is structured into subsystems, which contain objects that are functionally dependent on each other. The goal is to have objects with high coupling among each other in the same subsystem, whereas objects that are weakly coupled are in different subsystems. A subsystem can be considered a composite or aggregate object that contains the simple objects that compose that subsystem.

5.5.3. Concurrent Collaboration Diagrams

In the UML notation, an active object or task is depicted using a thick outline for the object box. An active object has its own thread of control and executes concurrently with other objects. This is in contrast to a passive object, which does not have a thread of control.

A passive object only executes when another object (active or passive) invokes one of its operations. In this paper, we refer to an active object as a task and a passive object as an object. Tasks are depicted on *concurrent collaboration diagrams,* which depict the concurrency aspects of the system [Douglass 2004]. On a concurrent collaboration diagram, a task is depicted as a box with thick black lines, whereas a passive object is depicted as a box with thin black lines. In addition, decisions are made about the type of message communication between tasks: asynchronous or synchronous, with or without reply.

5.5.4. Architectural Design of Distributed Real-Time Systems

Distributed real-time systems execute on geographically distributed nodes supported by a local or wide area network. With COMET, a distributed real-time system is structured into distributed subsystems, where a subsystem is designed as a configurable component and corresponds to a logical node. A subsystem component is defined as a collection of concurrent tasks executing on one logical node. As component subsystems potentially reside on different nodes, all communication between

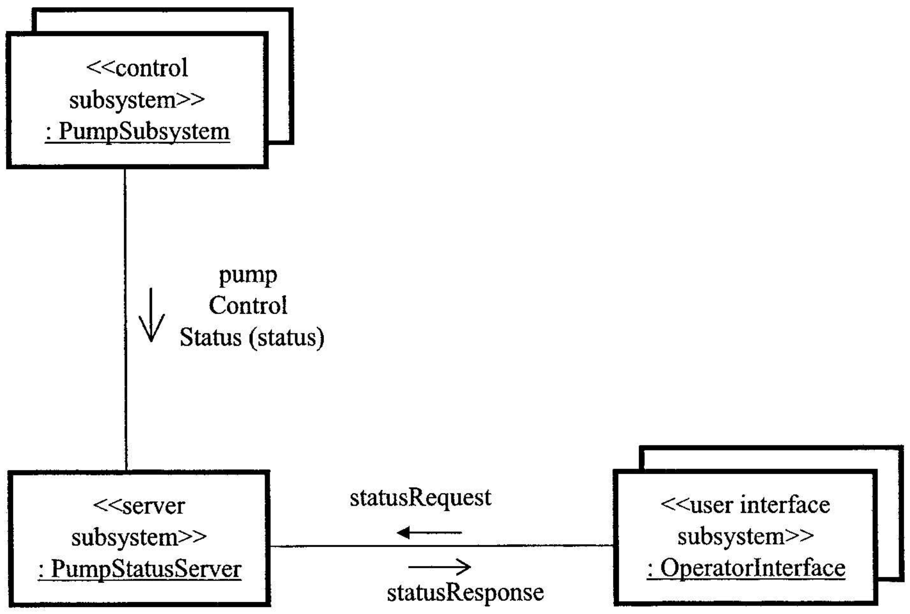

Figure 5. Distributed software architecture.

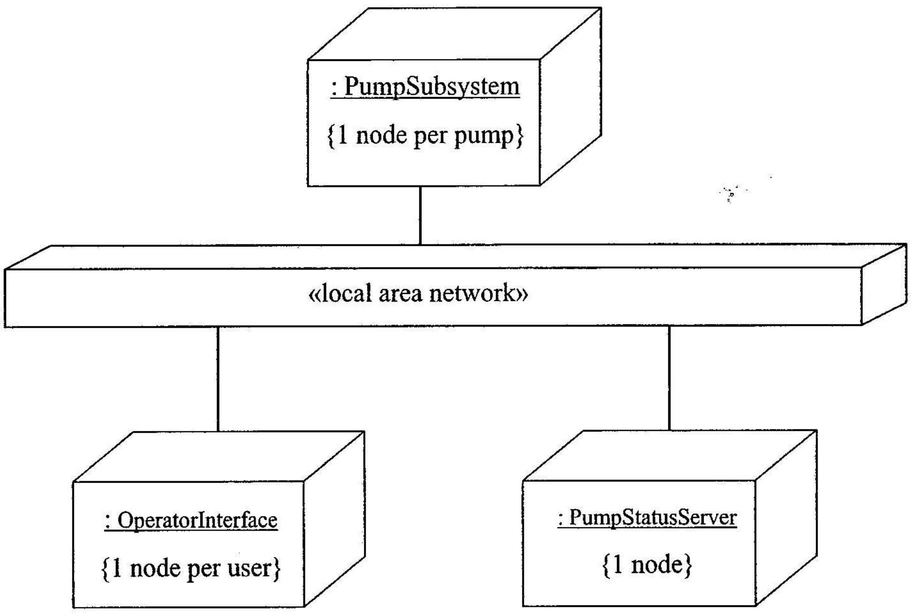

Figure 6. Distributed system configuration.

component subsystems must be restricted to message communication. Tasks in different subsystems may communicate with each other using several different types of message communication (Figure 5) including asynchronous communication, synchronous communication, client/server communication, group communication, brokered communication, and negotiated communication. The configuration of the distributed real-time system is depicted on a deployment diagram, as shown in Figure 6.

5.5.5. Task Structuring

During the task structuring phase, each subsystem is structured into concurrent tasks and the task interfaces are defined. Task structuring criteria are provided to assist in mapping an object-oriented analysis model of the system to a concurrent tasking architecture. Following the approach used for object structuring, stereotypes are used to depict the different kinds of tasks. Stereotypes are also used to depict the different kinds of devices the tasks interface with. During task structuring, if an object in the analysis model is determined to be active, then it is categorized further to show its task characteristics. For example, an active «I/O device interface» object is considered a task and categorized as one of the following: an «asynchronous I/O device interface» task, a «periodic I/O device interface» task, a «passive I/O device interface» task, or a «resource monitor» task. Similarly, an «external input device» is classified, depending on its characteristics, into an «asynchronous input device» or «passive input device».

An asynchronous I/O device interface task is needed when there is an asynchronous I/O device with which the system has to interface. For each asynchronous I/O device, there needs to be an asynchronous I/O device interface task to interface with it. The asynchronous I/O device interface task is activated by an interrupt from the asynchronous device.

Whereas an asynchronous I/O device interface task deals with an asynchronous I/O device, a periodic I/O device interface task deals with a passive I/O device, in which the device is polled on a regular basis. In this situation, the activation of the task is periodic but its function is I/O related. The periodic I/O device interface task is activated by a timer event, performs an I/O operation, and then waits for the next timer event.

An example of a task architecture for the pump monitoring and control system is given in Figure 7.

5.5.6. Detailed Software Design

In this step, the internals of composite tasks that contain nested objects are designed, detailed task synchronization issues are addressed, connector classes are designed that encapsulate the details of intertask communication, and each task's internal event sequencing logic is defined. An example of the detailed design of a composite task is given in Figure 8.

If a passive class is accessed by more than one task, then the class's operations must synchronize the access to the data it encapsulates. Synchronization is achieved using the mutual exclusion or multiple readers and writers algorithms [Bacon 1997].

Connector classes encapsulate the details of intertask communication, such as loosely and tightly coupled message communication. Some concurrent programming languages such as Ada and Java provide mechanisms for intertask communication

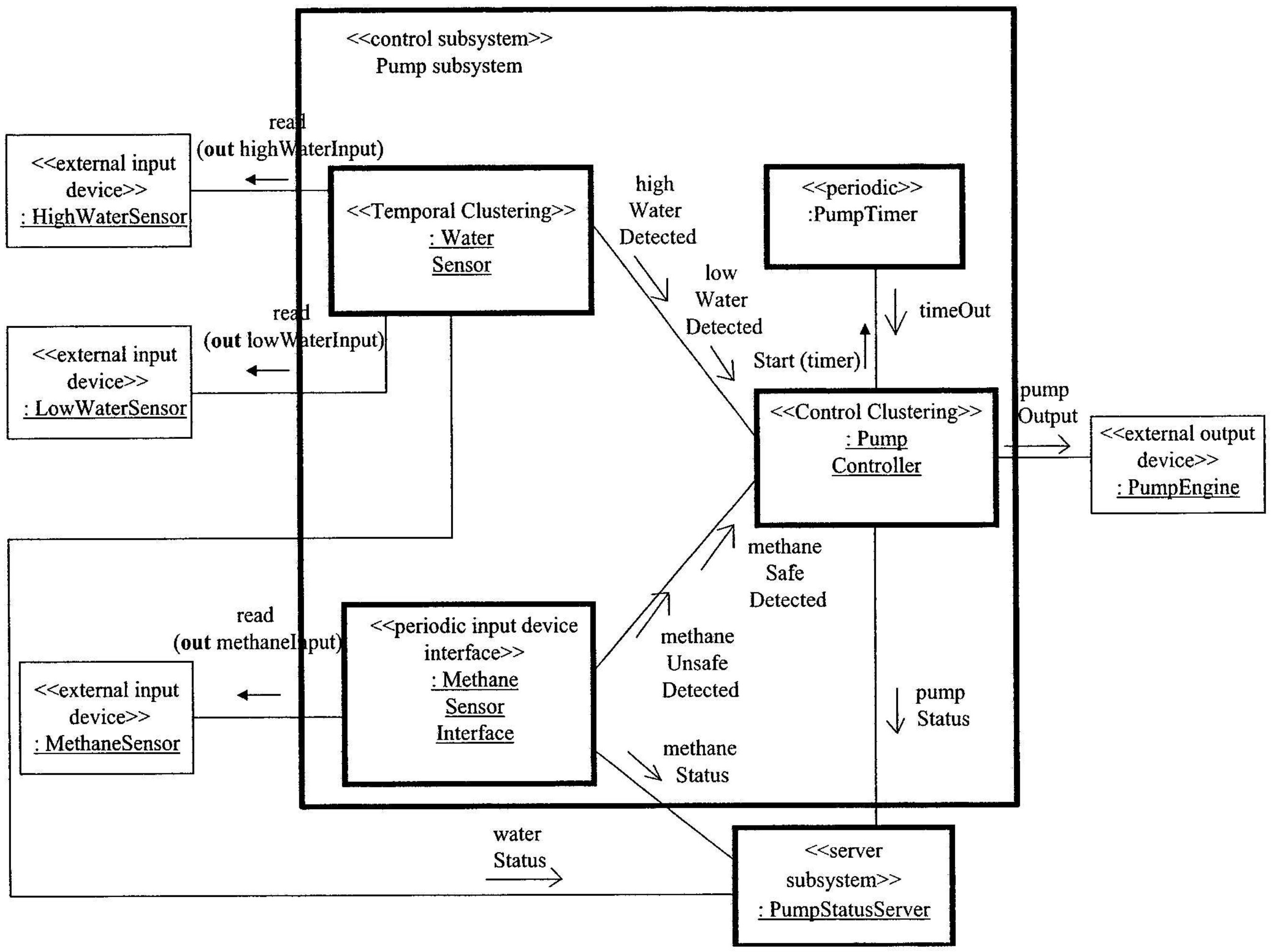

Figure 7. Pump subsystem, task architecture.

and synchronization. Neither of these languages supports loosely coupled message communication. In order to provide this capability, it is necessary to design a message queue connector class, which encapsulates a message queue and provides operations to access the queue. A connector is designed using a monitor, which combines the concepts of information hiding and task synchronization [Bacon 1997, Magee and Kramer 1999]. These monitors are used in a single processor or multiprocessor system with shared memory. Connectors may be designed to handle loosely coupled message communication, tightly coupled message communication without reply, and tightly coupled message communication with reply.

6. PERFORMANCE ANALYSIS OF REAL-TIME DESIGNS

Performance analysis of software designs is particularly important for real-time systems. The consequences of a real-time system failing to meet a deadline can be catastrophic.

The quantitative analysis of a real-time system design allows the early detection of potential performance problems. The analysis is for the software design conceptually executing on a given hardware configuration with a given external workload applied to it. Early detection of potential performance problems allows alternative software designs and hardware configurations to be investigated.

In COMET, performance analysis of software designs is achieved by applying *realtime scheduling* theory. *Real-time scheduling* is an approach that is particularly appropriate for hard real-time systems that have deadlines that must be met [Gomaa 1993, SEI 1993]. With this approach, the real-time design is analyzed to determine whether it can meet its deadlines.

A second approach for analyzing the performance of a design is to use *event sequence analysis* and to integrate this with *real-time scheduling* theory. Event sequence analysis considers scenarios of task collaborations and annotates them with the timing parameters for each of the tasks participating in each collaboration, in addition to system overhead for inter-object

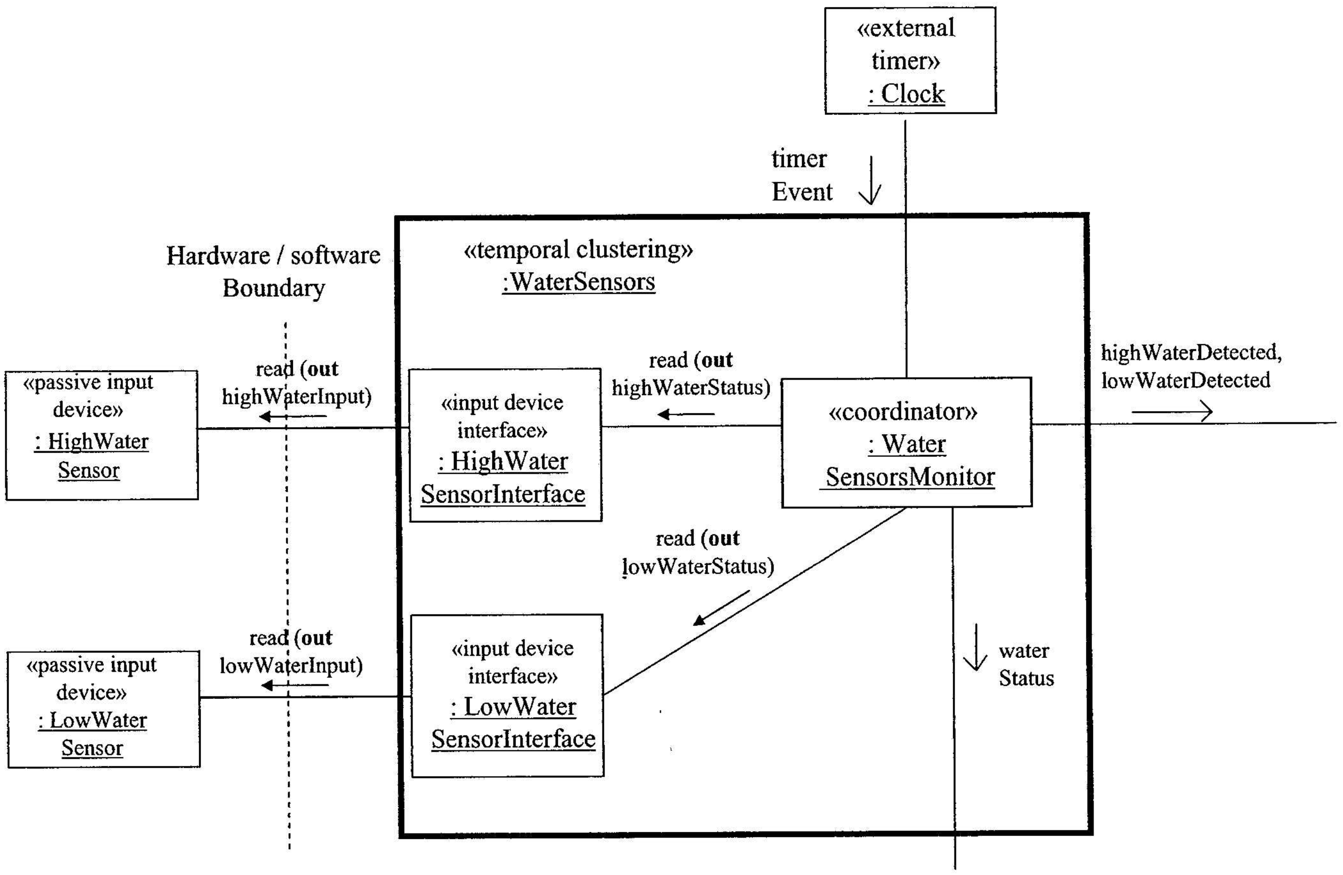
«external timer» : Clock
timer Event
Hardware / software Boundary
«temporal clustering» :WaterSensors
«passive input device» : HighWater Sensor
read (out highWaterInput)
«input device interface» : HighWater SensorInterface
read (out highWaterStatus)
«coordinator» : Water SensorsMonitor
highWaterDetected, lowWaterDetected
read (out lowWaterStatus)
«passive input device» : LowWater Sensor
read (out lowWaterInput)
«input device interface» : LowWater SensorInterface
water Status

Figure 8. Water sensors, temporal clustering with nested device interface objects.

communication and context switching. The equivalent period for the active objects in the collaboration is the minimum inter-arrival time of the external event that initiates the collaboration.

7. CONCLUSIONS

This paper has described concepts and methods for concurrent and real-time systems design. The concept of a concurrent task, as well as the basic mechanisms for intertask communication and synchronization, have been described.

When designing concurrent and real-time systems, it is essential to blend object-oriented concepts with the concepts of concurrent processing. This paper has described some of the key aspects of the COMET method for designing concurrent and real-time systems, which integrates object-oriented and concurrent processing concepts and uses the UML notation.

With the proliferation of low-cost workstations and personal computers operating in a networked environment, the interest in designing concurrent systems, particularly real-time and distributed systems, is likely to grow rapidly in the next few years. Furthermore, with the growing need for reusable designs, design methods for software product lines are likely to be of increasing importance for future real-time systems [Gomaa 2004].

BIBLIOGRAPHY

Awad, M., Kuusela, J., and Ziegler, J., *Object-Oriented Technology for Real-Time Systems,* Prentice-Hall, 1996.

Bacon, J., *Concurrent Systems,* 2nd ed., Addison-Wesley, 1997.

Barnes, J., *Programming in Ada 95,* Addison-Wesley, Reading MA, 1995.

Booch, G., *Object-Oriented Analysis and Design with Applications,* Addison-Wesley, Reading MA, 1994.

Booch, G., Rumbaugh, J., and Jacobson, I. *The Unified Modeling Language User Guide,* Addison-Wesley, Reading MA, 1999.

Buhr, R. J. A., and Casselman, R. S., *Use Case Maps for Object-Oriented Systems,* Prentice-Hall, 1996.

Coad, P., and Yourdon, E., *Object-Oriented Analysis,* Prentice-Hall 1991.

Douglass, B. P., *Doing Hard Time: UML, Objects, Frameworks, and Patterns in Real-Time Software Development,* Addison-Wesley, Reading MA, 1999.

Douglass, B. P., *Real-Time UML,* 3rd ed., Addison-Wesley, Reading MA, 2004

Fowler, M., and Scott, K., *UML Distilled,* 3rd ed., Addison-Wesley, Reading MA, 2004.

Gomaa, H., "A Software Design Method for Real Time Systems," *Communications ACM, 27,* 9, September 1984.

Gomaa, H., *Software Design Methods for Concurrent and Real-Time Systems,* Addison-Wesley, Reading MA, 1993.

Gomaa, H., *Designing Concurrent, Distributed, and Real-Time Applications with UML,* Addison-Wesley, Reading MA, 2000.

Gomaa, H., *Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures,* Addison-Wesley, Reading MA, 2004.

Harel, D., "On Visual Formalisms," *CACM, 31,* 5 (May 1988), 514–530.

Harel, D., and Gary, E., "Executable Object Modeling with Statecharts," in *Proceedings of 18th International Conference on Software Engineering,* Berlin, March 1996

Harel, D., and Politi, M., *Modeling Reactive Systems with Statecharts,* McGraw-Hill, 1998.

Hatley D., and Pirbhai, I., *Strategies for Real Time System Specification,* Dorset House, 1988.

Jackson, M., *System Development,* Prentice-Hall, 1983.

Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, Reading MA, 1992.

Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process,* Addison-Wesley, Reading MA, 1999.

Magee, J., and Kramer, J., *Concurrency: State Models and Java Programs,* Wiley, 1999.

Parnas, D., Clements, P., and Weiss, D., "The Modular Structure of Complex Systems," in *Proceedings of Seventh IEEE International Conference on Software Engineering,* Orlando, Florida, March 1984.

Rosenberg, D., and Scott, K., *Use Case Driven Object Modeling with UML,* Addison-Wesley, Reading MA, 1999.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

Rumbaugh, J., Booch, G., and Jacobson, I., *The Unified Modeling Language Reference Manual,* Addison-Wesley, Reading MA, 1999.

Schneider, G., and Winters, J. P., *Applying Use Cases: A Practical Guide,* Addison-Wesley, Reading MA, 1998.

SEI—Carnegie Mellon University Software Engineering Institute, *A Practioner's Handbook for Real-Time Analysis—Guide to Rate Monotonic Analysis for Real-Time Systems,* Kluwer Academic Publishers, Boston, 1993.

Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling,* Wiley, 1994.

Selic, B., Gullekson, G., and Ward, P., *Real-Time Object-Oriented Modeling,* Wiley, 1994.

Shaw M., and Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline,* Prentice-Hall, 1996.

Shlaer, S., and Mellor, S., *Object Oriented Systems Analysis,* Prentice-Hall, 1988.

Silberschatz, A., and Galvin, P., *Operating System Concepts,* 5th ed., Addison-Wesley, 1998.

Simpson, H., and Jackson, K., "Process Synchronization in MASCOT," *The Computer Journal, 17,* 4, 1979.

Simpson, H., "The MASCOT Method," *IEE/BCS Software Engineering Journal, 1*(3), 1986, 103–120.

Tanenbaum, A. S., *Modern Operating Systems,* Prentice-Hall, 1992.

Ward, P., and Mellor, S., *Structured Development for Real-Time Systems,* Vols. 1, 2 & 3, Yourdon Press, 1985.

Yourdon, E., *Modern Structured Analysis,* Prentice-Hall, 1989.